DeepQL: A Query Language for Deep Neural Networks

Neal Dawson-Elli
Ph. D Student, Chemical Engineering
University of Washington, Seattle

Ryan Stoddard
Ph. D Student, Chemical Engineering
University of Washington, Seattle

(Dated: February 23, 2018)

Abstract:

Deep Neural Networks (DNNs) have emerged as an extremely powerful tool for classification and prediction of complex processes with incredible accuracy, yet revealing the interactions within a DNN is challenging. We develop "DeepQL", which is a query language for interfacing with a DNN model. With DeepQL, one can answer complex questions about hypothetical scenarios of a DNN model in a database-style language. For example, with a model predicting mortgage approval could be queried with DeepQL to answer, "is there anyone from Texas who would have had their mortgage approval outcome change had their race been listed differently"? We demonstrate DeepQL on an example dataset predicting H1B Visa case status, which shows promising utility for several example queries. Methods, implementation, and future work is discussed.

Introduction

A deep neural network (DNN) trained with an appropriate dataset can model output such as likely hood of getting a mortgage approved, value of a home, or a person's chances of getting heart disease. Typically, a DNN is deployed to take input variables (such as a person's income, age, credit history, mortgage amount, etc.) and predict an output (mortgage approved or not). However, it is not trivial to query the DNN output with complex questions such as is there a person such that changing their race would change the outcome of mortgage decision or in which zip codes can adding windows increase home value by more than 10.

We couldn't find prior work attempting to interface a SQL-style language with a DNN. However, developing methods to reveal a deeper understanding of DNNs has been largely pursued. For example, Yosinski *et. al* and Zeiler *et. al* use interesting visualization methods to show connections in DNNs [1-2]

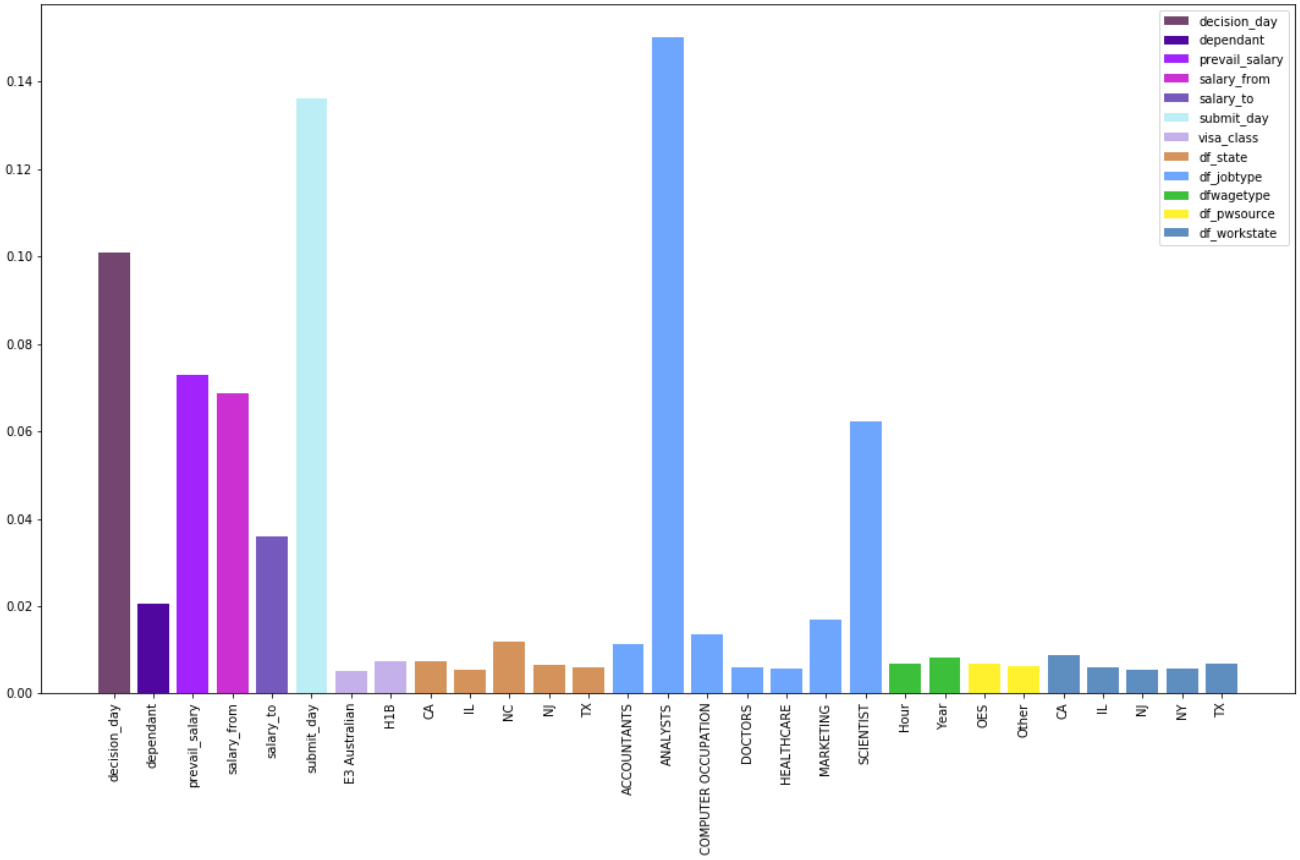
Methods

A. Machine Learning

The machine learning portion of the project has been implemented using the Keras library in Python. It consists of a Deep Neural Network (DNN) of depth 4 with dimensions [128, 128,128,128,4] with a dropout of 0.4 applied to each layer. The focus of the project is the interface between the query language and the DNN, so while some effort has been put into achieving a good score, minimizing the loss on the training set was not of paramount importance. What is more interesting is examining queries where values are changed and examining when the

model determines that the output will be different as a result. In order to get a better feel for the sensitivity of the model to different parameters, a Random Forest was also trained using the Sci-Kit Learn package in Python. The main advantage of using a Random Forest is that the trained model has a built-in sensitivity analysis, which scores the input features by the number of selections that were made based on each feature. Labeled 'Feature Importance', this metric is useful for getting a sense of whether or not modifying that input will likely result in a changed output. The feature importance (DeepQL/deepql/model_train/feature_importances.txt) give us an idea of the relative importance of the features, which allows us to design test queries that will cause changes in model prediction to help us evaluate if our language works as expected. Shown in Figure 1 below, it becomes apparent that of the 191 input features, only a handful are commonly used to predict values.

Figure 1. Feature importance of the model inputs, as seen by a Random Forest. The decision



day, submission day, and analyst and scientist job types are the most important, meaning that they are the most predictive features in the data set.

B. Dataset

Our initial model is done using the H1B Dataset, but an effort is being made to make the results general enough that a new data set could be adapted without too much difficulty. We are organizing our project in a GitHub repository, in hopes that our language can be deployed and used to interface with datasets outside of the H1B dataset. The H1B Dataset had unique challenges. One challenge is that it is an imbalanced dataset. The output parameter is the "case

status" of an H1B visa application, which can have 1 of 4 possible outcomes, "Certified", Withdrawn", "Certified-Withdrawn", or "Denied" – so the task is to correctly predict case status (one of the four possible outcomes) using the input data. The dataset is imbalanced since there is not an even distribution in outcomes, ~87% are Certified while <1% are Denied. Also, the dataset is weighted towards categorical variables, which we expanded out in our coding of the variables resulting in 191 input columns (with > 0.5 million rows). Although the DNN model development wasn't the focus of this project, we were able to get decent performance with 97.3% accuracy. Although there is still some miss-classifications (especially for denied), our model provides a nice vehicle to execute and test our query language. In order to try to make sense of the input space, we employed t-SNE dimensionality reduction, as shown in Figure 2 [3]. The technique is a variation of Stochastic Neighbor Embedding, and is implemented in the python package Sci-Kit Learn.

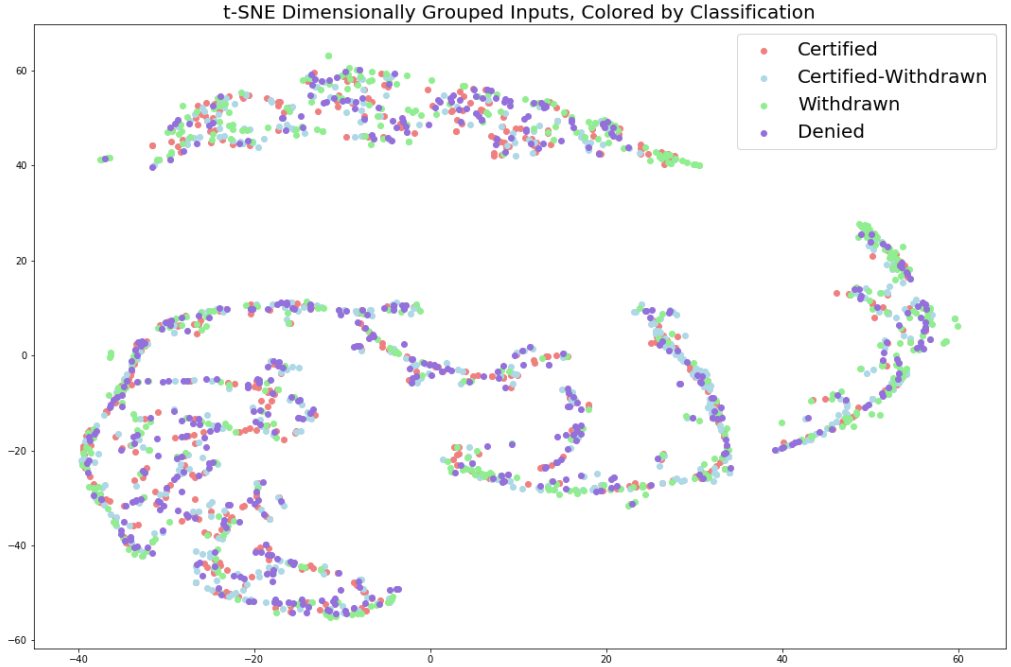


Figure 2. Visualization of a 2d reduced-parameter space via t-SNE dimensionality reduction. We were hoping to see clusters that correlated with the application being certified or withdrawn, but the fairly complex pattern seems to be more or less uniformly distributed across all 4 classes.

C. Implementation

At a high level, our implementation is straightforward, and is shown in Figure 3. When DeepQL is installed, the model is trained and the input data is loaded into an SQLite database (or the keras model and SQL database is downloaded). Note that for this dataset, training the model will take considerable time without keras-GPU capability. At runtime, the query is first divided into an SQL component and a DeepQL component. We use the token "WHAT IF", which does not appear in SQLite. The SQL component is sent to SQLite to select a subset of the training data to modify. The DeepQL component is sent to a parser to determine what specifically to modify in the training data. That information is sent to a wrapper, which modifies the raw table with desired modifications, then preprocesses it into an array in the same format as the DNN model was trained on. Then this is sent to model and results are returned. DeepQL takes inputs

of the form: **'SQL-query WHAT IF 'COLLUMN_NAME' = VALUE [TIMES] [AND ...] [DIFERENCE [OLD_RESULT TO NEW_RESULT]]'**. TIMES is an optional token for dealing with numerical data (sending a factor to multiply current value instead of providing the new value). DIFFERENCE is a token for returning only those values where the prediction has changed based on the modifications. If the optional OLD_RESULT TO NEW_RESULT information is provided after DIFFERENCES, then only the specific cases where the predicted output changes from OLD_RESULT to NEW_RESULT are returned. Using AND can allow for searching the Cartesian product of multiple modifications to see in which cases the predicted output is modified.

.

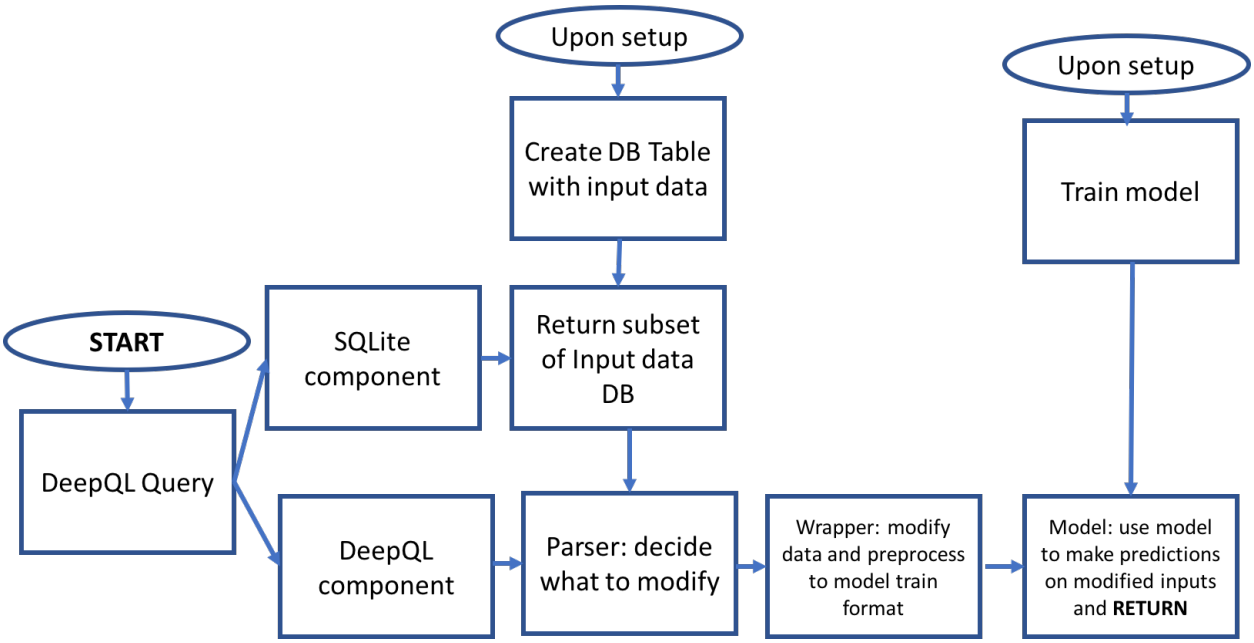


Figure 3. *Schematic of DeepQL implementation*

Now we will walk through an example of a DeepQL query. Consider the following query input:

"select * from BASE where jobName = 'EDUCATION' limit 1000 WHAT IF SOC_NAME = SCIENTIST DIFFERENCE"

The token WHAT IF divides into SQL component and DeepQL component. So, the shorter string “select * from BASE where jobName = 'EDUCATION' limit 1000” is sent to the SQLite Table “BASE” where the 500,000 input data resides. The remaining part of the string “WHAT IF SOC_NAME = SCIENTIST DIFFERENCE” is sent to parser, which finds that the desired change is to change the SOC_NAME (which is what the training data calls the jobName) to “scientist”. The DIFFERENCE token signals to only return entries where the prediction changes. This query finds the first 1000 entries with “education” jobname, switches the jobname to “scientist”, and returns cases in which the modeled case status is different. In addition to directly modifying text fields, we have also implemented a TIMES token, which allows for the direct multiplication of the continuous inputs, which are stored as float values. For example, the token “TIMES = 2” would multiply the previously selected field by a factor of 2. Additional modified fields can be strung together using the AND token, which returns the Cartesian product of the requested modifications. For example, “IF SOC_NAME = SCIENTIST AND

PREVAILING_WAGE TIMES = 2" would provide every combination of the original SOC_NAME, the modified SOC_NAME, the original PREVAILING_WAGE, and the modified PREVAILING_WAGE. These would then be fed into the neural network, and the fields which changed could be returned by submitting DIFFERENCE. In addition to simply selecting the fields that were different, we also extended this to include the ability to specify what difference the query is interested in – for example, only returning results which go from CERTIFIED to DENIED, or vice-versa. In this way, it is possible to specify exactly the desired results. Everything is implemented in python using common libraries (numpy, pandas, scikitlearn, and keras), with sqlite3 to interface python with SQLite.

D. Results

We considered several example queries to test the performance of DeepQL. The example query discussed above:

"select * from BASE where jobName = 'EDUCATION' limit 1000 IF SOC_NAME = SCIENTIST DIFFERENCE"

returns 1000 results from BASE (indicating at least 1000 entries with jobname "education"), and the returns 30 rows with different predictions. This indicate that of the 1000 results returned, 30 will have a different outcome if the jobname is changed from EDUCATION to SCIENTIST. Some of the changes are from Certified to Withdrawn, some from Certified to Denied, and one from Denied to Withdrawn (none from Denied to Certified!). Next, we tried a test query for dealing with numerical inputs. The following query

"select * from BASE where wage > 60000 IF PREVAILING_WAGE = 0.5 TIMES DIFFERENCE"

returns 361,947 results from SQLite query (cases of the ~0.5M that have wage >60000) and returns 32 rows of people who would have had their case status change if their salary was half of the original salary. These results are mostly a mix of certified to denied and certified to withdrawn. To test the functionality of "AND" as well as "OLD_RESULT TO NEW_RESULT", we tested a few more sophisticated queries.

"select * from BASE where jobName = 'EDUCATION' WHAT IF SOC_NAME = SCIENTIST AND H-1B_DEPENDENT = N DIFFERENCE DENIED TO CERTIFIED"

returns 12,370 entries from SQL database (number of entries with 'EDUCATION' jobname), yet returns 0 results from DeepQL query, indicating that in no cases any combination of changing education to scientist with Y or N H1B dependent yields a predicited change in case status from Denied to Certified. Note that we wrote this example query to highlight semantics of our AND functionality – we did not specify "where H1B_dependent = X" in the SQL part, meaning that the returned rows do not care about initial H1B dependent status. The AND adds the cartesien product of changes (for two changes, old-new, new-old, new-new), regardless of what the old entry contained. The following query is very similar but gives very different results:

"select * from BASE where jobName = 'SCIENTIST' WHAT IF SOC_NAME = EDUCATION AND H-1B_DEPENDENT = N DIFFERENCE DENIED TO CERTIFIED"

returns 17,939 rows from SQL part and 1116 results where the changes invoke a case status change from Denied to Certified. This two queries indicate that there is an "edge" surrounding jobname, where Scientist is more likely to be denied and education is more likely to be certified.

Our queries are currently executed in the DeepQL/deepql/call/call.ipynb file. In summary, we have made some progress and are getting results to simple example queries similar to what we set out to accomplish. We would like to expand the functionality of the DeepQL queries in future work.

E. Future Work

Potential future work includes expanding upon the functionality of DeepQL, as well as refactoring our code and improving the repo structure, which would allow others to import it as a module or pip-install the project as a package. Specifically, we would like to implement a wildcard (*) difference for categorical variables, which would require storing a list of the unique variables present in each categorical feature. While this would allow for more interesting queries – equivalent to "is there ANY job that would result in changing from certified to denied" – allowing this option on multiple inputs would cause the number of combinations to become computationally intractable relatively quickly.

We would also like to implement some sort of sampling-based techniques for reducing the size of the cartesian product in instances like these, but that would result in identical queries returning different results, which is undesired. It would also be possible that, for large queries like this, a form of pipelining could be implemented which would allow for reduced RAM requirements in the machines attempting to use DeepQL. For instance, the current implementation relies upon loading the entire SQL query into memory and making all of the modifications in memory. While future implementations would also likely load the entire SQL output into memory, it would be possible to chunk the cartesian product into a more manageable sample size and feed those chunks into the neural network, filtering the output, and returning the full output when completed. This would likely not make query process any faster, but it would spare the burden of requiring a large amount of RAM to run.

Perhaps the most difficult future work would be generalizing the preprocessing function, which would allow for this method to be extended to additional datasets. In general, this requires creating a feature vector for each categorical value and transforming the continuous variables from their current value to a value on the order of [0~1]. This expands the feature input space, but the scaling allows for optimal functioning of the neural network. We would also like to write tests for our code, and possibly integrate TravisCI, a tool that runs the created tests each time a push is made to Github. Currently, our implementation has very little error catching, and will not necessarily fail elegantly if a categorical value is given that does not exist in the data. Writing tests would require handling these and could potentially include implementing SQL-like suggested modifications – "you entered 'ENGNEER', did you mean 'ENGINEER'?" for example. Python has many useful tools for finding similarity in strings that may be possible to leverage towards this end.

In summary, we have made some progress in a deep and interesting research area. Our language only scratches the surface of what is possible – we only consider modifications on existing cases, not the essentially infinite cases outside of the dataset. DeepQL can be used to ask complex queries of DNN output and get invaluable insight on DNN models.

F. Division of Labor

Typically, in larger-scale projects, tasks will be divided up and seen to completion by a single member of the team. While that was somewhat the case with this project, it was manageable for both parties to contribute to all aspects. Ryan began with preprocessing the data

and creating a small neural network, and went on to begin the implementation of the token system, pioneering the work related to replacing categorical variables and filtering the returned values, including the implementation of the difference token and the specific types of differences. Neal focused on storing the data in SQlite and spent most of his effort on improving the output from the neural network, as well as implementing the times token and making various optimizations and implementing some error handling.

References:

1. arXiv:1506.06579v1
2. Zeiler M.D., Fergus R. (2014) Visualizing and Understanding Convolutional Networks. In: Fleet D., Pajdla T., Schiele B., Tuytelaars T. (eds) Computer Vision – ECCV 2014. ECCV 2014. Lecture Notes in Computer Science, vol 8689. Springer, Cham
3. L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," Journal of Machine Learning Research, vol. 9, no. Nov, pp. 2579–2605, 2008.